# <a name="main-heading"></a>Dolphin Coding Style & Licensing

If you make any contributions to Dolphin after December 1st, 2014, you are agreeing that any code you have contributed will be licensed under the GNU GPL version 2 (or any later version).

# <a name="main-section-overview"></a>Main sections

- [Introduction](#introduction)

- [C++ coding style and formatting](#cpp-coding-style-and-formatting)

- [C++ code-specific guidelines](#cpp-code-specific-guidelines)

- [Android and Java](#android-and-java)

# <a name="introduction"></a>Introduction

Summary:

- [Aims](#intro-aims)

- [Checking and fixing formatting issues](#intro-formatting-issues)

## <a name="intro-aims"></a>Aims

This guide is for developers who wish to contribute to the Dolphin codebase. It will detail how to properly style and format code to fit this project. This guide also offers suggestions on specific functions and other varia that may be used in code.

Following this guide and formatting your code as detailed will likely get your pull request merged much faster than if you don't (assuming the written code has no mistakes in itself).

This project uses clang-format (stable branch) to check for common style issues. In case of conflicts between this guide and clang-format rules, the latter should be followed instead of this guide.

## <a name="intro-formatting-issues"></a>Checking and fixing formatting issues

Windows users need to be careful about line endings. Windows users should configure git to checkout UNIX-style line endings to keep clang-format simple.

In most cases, clang-format can and \*\*should\*\* be used to automatically reformat code and solve most formatting issues.

- To run clang-format on all staged files:

```

git diff --cached --name-only | egrep '[.](cpp|h|mm)$' | xargs clang-format -i

```

- Formatting issues can be checked for before committing with a lint script that is included with the codebase. To enable it as a pre-commit hook (assuming you are in the repository root):

```

ln -s ../../Tools/lint.sh .git/hooks/pre-commit

```

- Alternatively, a custom git filter driver can be used to automatically and transparently reformat any changes:

```

git config filter.clang\_format.smudge 'cat'

git config filter.clang\_format.clean 'clang-format %f'

echo '/Source/Core/\*\*/\*.cpp filter=clang\_format' >> .git/info/attributes

echo '/Source/Core/\*\*/\*.h filter=clang\_format' >> .git/info/attributes

echo '/Source/Core/\*\*/\*.mm filter=clang\_format' >> .git/info/attributes

```

# <a name="cpp-coding-style-and-formatting"></a>C++ coding style and formatting

Summary:

- [General](#cpp-style-general)

- [Naming](#cpp-style-naming)

- [Conditionals](#cpp-style-conditionals)

- [Classes and structs](#cpp-style-classes-and-structs)

## <a name="cpp-style-general"></a>General

- Try to limit lines of code to a maximum of 100 characters.

- Note that this does not mean you should try and use all 100 characters every time you have the chance. Typically with well formatted code, you normally shouldn't hit a line count of anything over 80 or 90 characters.

- The indentation style we use is 2 spaces per level.

- The opening brace for namespaces, classes, functions, enums, structs, unions, conditionals, and loops go on the next line.

- With array initializer lists and lambda expressions it is OK to keep the brace on the same line.

- References and pointers have the ampersand or asterisk against the type name, not the variable name. Example: `int\* var`, not `int \*var`.

- Don't use multi-line comments (`/\* Comment text \*/`), use single-line comments (`// Comment text`) instead.

- Don't collapse single line conditional or loop bodies onto the same line as its header. Put it on the next line.

- Yes:

```c++

if (condition)

return 0;

while (var != 0)

var--;

```

- No:

```c++

if (condition) return 0;

while (var != 0) var--;

```

## <a name="cpp-style-naming"></a>Naming

- All class, enum, function, and struct names should be in upper CamelCase. If the name contains an abbreviation uppercase it.

- `class SomeClassName`

- `enum IPCCommandType`

- All compile time constants should be fully uppercased. With constants that have more than one word in them, use an underscore to separate them.

- `constexpr double PI = 3.14159;`

- `constexpr int MAX\_PATH = 260;`

- All variables should be lowercase with underscores separating the individual words in the name.

- `int this\_variable\_name;`

- Please do not use [Hungarian notation](http://en.wikipedia.org/wiki/Hungarian\_notation) prefixes with variables. The only exceptions to this are the variable prefixes below.

- Global variables – `g\_`

- Class variables – `m\_`

- Static variables – `s\_`

## <a name="cpp-style-conditionals"></a>Conditionals

- Do not leave `else` or `else if` conditions dangling unless the `if` condition lacks braces.

- Yes:

```c++

if (condition)

{

// code

}

else

{

// code

}

```

- Acceptable:

```c++

if (condition)

// code line

else

// code line

```

- No:

```c++

if (condition)

{

// code

}

else

// code line

```

## <a name="cpp-style-classes-and-structs"></a>Classes and structs

- If making a [POD](http://en.wikipedia.org/wiki/Plain\_Old\_Data\_Structures) type, use a `struct` for this. Use a `class` otherwise.

- Class layout should be in the order, `public`, `protected`, and then `private`.

- If one or more of these sections are not needed, then simply don't include them.

- For each of the above specified access levels, the contents of each should follow this given order: constructor, destructor, operator overloads, functions, then variables.

- When defining the variables, define `static` variables before the non-static ones.

```c++

class ExampleClass : public SomeParent

{

public:

ExampleClass(int x, int y);

int GetX() const;

int GetY() const;

protected:

virtual void SomeProtectedFunction() = 0;

static float s\_some\_variable;

private:

int m\_x;

int m\_y;

};

```

# <a name="cpp-code-specific-guidelines"></a>C++ code-specific guidelines

Summary:

- [General](#cpp-code-general)

- [Headers](#cpp-code-headers)

- [Loops](#cpp-code-loops)

- [Functions](#cpp-code-functions)

- [Classes and Structs](#cpp-code-classes-and-structs)

## <a name="cpp-code-general"></a>General

- The codebase currently uses C++17.

- Use the [nullptr](http://en.cppreference.com/w/cpp/language/nullptr) type over the macro `NULL`.

- If a [range-based for loop](http://en.cppreference.com/w/cpp/language/range-for) can be used instead of container iterators, use it.

- Obviously, try not to use `goto` unless you have a \*really\* good reason for it.

- If a compiler warning is found, please try and fix it.

- Try to avoid using raw pointers (pointers allocated with `new`) as much as possible. There are cases where using a raw pointer is unavoidable, and in these situations it is OK to use them. An example of this is functions from a C library that require them. In cases where it is avoidable, the STL usually has a means to solve this (`vector`, `unique\_ptr`, etc).

- Do not use the `auto` keyword everywhere. While it's nice that the type can be determined by the compiler, it cannot be resolved at 'readtime' by the developer as easily. Use auto only in cases where it is obvious what the type being assigned is (note: 'obvious' means not having to open other files or reading the header file). Some situations where it is appropriate to use `auto` is when iterating over a `std::map` container in a foreach loop, or to shorten the length of container iterator variable declarations.

- Do not use `using namespace [x];` in headers. Try not to use it at all if you can.

- The preferred form of the increment and decrement operator in for-loops is prefix-form (e.g. `++var`).

## <a name="cpp-code-headers"></a>Headers

- If a header is not necessary in a certain source file, remove them.

- If you find duplicate includes of a certain header, remove it.

- When declaring includes in a source file, make sure they follow the given pattern:

- The header for the source file

- Standard library headers

- System-specific headers (these should also likely be in an `#ifdef` block unless the source file itself is system-specific).

- Other Dolphin source file headers

- Each of the above header sections should also be in alphabetical order

- Project source file headers should be included in a way that is relative to the `[Dolphin Root]/Source/Core` directory.

- This project uses `#pragma once` as header guards.

## <a name="cpp-code-loops"></a>Loops

- If an infinite loop is required, do not use `for (;;)`, use `while (true)`.

- Empty-bodied loops should use braces after their header, not a semicolon.

- Yes: `while (condition) {}`

- No: `while (condition);`

- For do-while loops, place 'while' on the same line as the closing brackets

```c++

do

{

} while (false);

```

## <a name="cpp-code-functions"></a>Functions

- If a function parameter is a pointer or reference and its value or data isn't intended to be changed, please mark that parameter as `const`.

- Functions that specifically modify their parameters should have the respective parameter(s) marked as a pointer so that the variables being modified are syntaxically obvious.

- What not to do:

```c++

template<class T>

inline void Clamp(T& val, const T& min, const T& max)

{

if (val < min)

val = min;

else if (val > max)

val = max;

}

```

Example call: `Clamp(var, 1000, 5000);`

- What to do:

```c++

template<class T>

inline void Clamp(T\* val, const T& min, const T& max)

{

if (\*val < min)

\*val = min;

else if (\*val > max)

\*val = max;

}

```

Example call: `Clamp(&var, 1000, 5000);`

- Class member functions that you do not want to be overridden in inheriting classes should be marked with the `final` specifier.

```c++

class ClassName : ParentClass

{

public:

void Update() final;

};

```

- Overridden member functions that can also be inherited should be marked with the `override` specifier to make it easier to see which functions belong to the parent class.

```c++

class ClassName : ParentClass

{

public:

void Update() override;

};

```

## <a name="cpp-code-classes-and-structs"></a>Classes and structs

- Classes and structs that are not intended to be extended through inheritance should be marked with the `final` specifier.

```c++

class ClassName final : ParentClass

{

// Class definitions

};

```

# <a name="android-and-java"></a>Android and Java

The Android project is currently written in Java. If you are using Android Studio to contribute, you can import the project's code style from `code-style-java.jar`, located in `[Dolphin Root]/Source/Android`. Please organize imports before committing.